Lecture 7: Static ILP and branch prediction

• Topics: static speculation and branch prediction
  (Appendix G, Section 2.3)
Support for Speculation

• In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences

• However, we need hardware support
  ➢ to ensure that an exception is raised at the correct point
  ➢ to ensure that we do not violate memory dependences
Detecting Exceptions

• Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)

• For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss

• In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative

• Note that a speculative instruction needs a special opcode to indicate that it is speculative
Program-Terminate Exceptions

• When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register

• If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the core-dump happens two procedures later)

• Alternatively, an instruction (the sentinel) in the speculative instruction’s original location checks the register value and initiates recovery
Memory Dependence Detection

• If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

• When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

• If a store finds its address in the ALAT, it indicates that a violation occurred for that address

• A special instruction (the sentinel) in the load’s original location checks to see if the address had a violation and re-executes the load if necessary
Dynamic Vs. Static ILP

- Static ILP:
  + The compiler finds parallelism $\rightarrow$ no scoreboarding $\rightarrow$ higher clock speeds and lower power
  + Compiler knows what is next $\rightarrow$ better global schedule
  - Compiler can not react to dynamic events (cache misses)
  - Can not re-order instructions unless you provide hardware and extra instructions to detect violations (eats into the low complexity/power argument)
  - Static branch prediction is poor $\rightarrow$ even statically scheduled processors use hardware branch predictors
  - Building an optimizing compiler is easier said than done

- A comparison of the Alpha, Pentium 4, and Itanium (statically scheduled IA-64 architecture) shows that the Itanium is not much better in terms of performance, clock speed or power
Control Hazards

• In the 5-stage in-order processor: assume always taken or assume always not taken; if the branch goes the other way, squash mis-fetched instructions (momentarily, forget about branch delay slots)

• Modern in-order and out-of-order processors: dynamic branch prediction; instead of a default not-taken assumption, either predict not-taken, or predict taken-to-X, or predict taken-to-Y

• Branch predictor: a cache of recent branch outcomes
Pipeline without Branch Predictor

In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch
In the 5-stage pipeline, a branch completes in two cycles → If the branch went the wrong way, one incorrect instr is fetched → One stall cycle per incorrect branch
Branch Mispredict Penalty

• Assume: no data or structural hazards; only control hazards; every 5\text{th} instruction is a branch; branch predictor accuracy is 90%

• Slowdown = 1 / (1 + stalls per instruction)

• Stalls per instruction = \% branches x \%mispreds x penalty
  = 20\% x 10\% x 1
  = 0.02

• Slowdown = 1/1.02 ; if penalty = 20, slowdown = 1/1.4
1-Bit Prediction

• For each branch, keep track of what happened last time and use that outcome as the prediction

• What are prediction accuracies for branches 1 and 2 below:

```c
while (1) {
    for (i=0;i<10;i++) {                     branch-1
        ...
    }
    for (j=0;j<20;j++) {                     branch-2
        ...
    }
}
```
2-Bit Prediction

• For each branch, maintain a 2-bit saturating counter:
  if the branch is taken: counter = min(3,counter+1)
  if the branch is not taken: counter = max(0,counter-1)

• If (counter >= 2), predict taken, else predict not taken

• Advantage: a few atypical branches will not influence the prediction (a better measure of “the common case”)

• Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)

• Can be easily extended to N-bits (in most processors, N=2)
Title

• Bullet